



LAWRENCE  
LIVERMORE  
NATIONAL  
LABORATORY

# Design and Optimization of a Metagenomics Analysis Workflow for NVRAM

S. Ames, J. Allen, D. Hysom, S. Lloyd, M.  
Gokhale

April 30, 2013

IPDPS 14  
Phoenix, AZ, United States  
May 19, 2014 through May 23, 2014

## **Disclaimer**

---

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

# Design and Optimization of a Metagenomics Analysis Workflow for NVRAM

Sasha Ames, Jonathan E. Allen, David A. Hysom, G. Scott Lloyd and Maya B. Gokhale  
Lawrence Livermore National Laboratory  
Contact: ames4@llnl.gov

## ABSTRACT

Metagenomic analysis, the study of microbial communities found in environmental samples, presents considerable challenges in quantity of data and computational cost. We present a novel metagenomic analysis pipeline that leverages emerging large address space compute nodes with NVRAM to hold a searchable, memory-mapped “k-mer” database of all known genomes and their taxonomic lineage. We describe challenges to creating the many hundred gigabyte-sized databases and describe database organization optimizations that enable our Livermore Metagenomic Analysis Toolkit (LMAT) software to effectively query the k-mer key-value store, which resides in high performance flash storage, as if fully in memory.

To make database creation tractable, we have designed, implemented, and evaluated an optimized ingest pipeline. To optimize query performance for the database, we present a two-level index scheme that yields speedups of  $8.4 \times -74\times$  over a conventional hash table index. LMAT, including the ingest pipeline, is available as open source at SourceForge.

## 1. INTRODUCTION

Metagenomics is concerned with characterizing and analyzing microbial communities present in diverse natural environments. Metagenomics sequencing has emerged as a powerful genetic survey tool used in research for generating a far more unbiased and detailed description of a biological sample than has ever been possible. It is the nature of a metagenomic input set that the organisms it contains are diverse and largely unknown, and analysis of such data sets represents a data intensive analysis problem that challenges conventional computing approaches. Present approaches to metagenomic analysis rely on sequence alignment tools that match each genetic fragment (“read”) to a part of each reference sequence and report a summary of the top reference matches. However, with rapidly growing sequencer throughput, alignment-based approaches are facing severe scaling limitations, even with high CPU core counts. To enable

near real-time analysis of metagenomic datasets at field sites co-located with the sequencer, we have developed a novel alignment-free approach, the Livermore Metagenomic Analysis Toolkit (LMAT) that exploits large (persistent) memory to store a searchable database of k-mers, all k-length sequences from a reference set of genomes [2]. By transferring the computational load to an offline database generation phase, we transform the analysis problem to parallel search of a read-only database. Our approach has been shown to perform favorably in accuracy and speed to competing alignment-based techniques [2]. The source code is available on Sourceforge ([1]).

Our approach leverages emerging large address space compute nodes with NVRAM to hold a searchable k-mer database of all known genomes and their taxonomic lineage. The database resides in a file in NVRAM, which is mapped into the address space of the application, allowing the application to access data structures directly as if in memory. This approach anticipates future memory hierarchies incorporating NVRAM as a high capacity, high latency last-level memory. We show that even with today’s NAND flash arrays, we achieve excellent query processing rates. In this paper we describe the workflow pipeline to generate the searchable k-mer database; evaluate database organization optimizations; analyze speed performance for database query using real metagenomic query sets. The full LMAT database reaches almost 500GB in size. It is organized as a key-value store in which the data stored with each k-mer key records alternative taxonomic lineage for the k-mer. To make database creation tractable for our reference database, we parallelize generating the k-mers of reference genomes. We evaluate alternative key-value store organizations, and have developed a two level index optimized for Flash storage. During query, the database is memory-mapped into the address space of the query process, and we use a custom mmap handler optimized for data intensive applications [17]. The optimized index gives a  $8.4 \times -74\times$  speedup over a conventional hash table index. The speed performance of our approach is evaluated using 0.5-2.5 GB query sets.

Our contributions are as follows:

- demonstrate a scalable new approach to generating large, memory-mapped searchable k-mer databases, starting from NCBI reference genome sets,
- evaluate alternative approaches to the database ingest

pipeline, including parallelization on a conventional cluster and on a single large memory node,

- design a two-level index data structure uniquely tuned to flash array access,
- demonstrate speed improvement on query of more than an order of magnitude over conventional key-value store organizations.

## 2. LMAT SEARCH AND CLASSIFICATION

LMAT identifies organisms in a metagenomic sample by matching k-mers in the reads in the sample with k-mers in a searchable database. If a set of k-mers from a read match k-mers from a particular reference sequence, then there is some likelihood that the read may come from the species or strain of the reference. On the other hand, a read may contain k-mers that map to a diverse group of organisms that have a common ancestor in the biological taxonomy at the level of genus, family, or higher. The LMAT classification algorithm is rank-flexible, attempting to classify a read at the lowest level possible of the taxonomy tree. LMAT classification uses threshold values for the classifier, so there must be a strong enough signal for a particular match to make a “call” for a particular taxonomy. For rank flexibility, for instance, if there is a match for a particular strain (lowest rank) but it is not enough to pass the threshold, it might be that there is enough for a species (next rank up the NCBI taxonomy hierarchy) level match, and so on. LMAT uses a searchable index of k-mers that map to constituent taxonomic identifiers for the organisms and higher common ranks when appropriate.

Other approaches that map reads to reference sequences using k-mers rely on a small subset of k-mers to represent the database in order to minimize the challenges of searching a large reference database. A unique feature of LMAT is its use of all k-mers in the reference database to improve detection sensitivity. Since the number of k-mers in the current reference database is approaching 20 billion and will continue to grow as new organisms are sequenced, new approaches are needed to efficiently manage and retrieve information from the reference database. One key challenge is to identify, store and manage the billions of k-mers in a way that avoids duplicates, allows for adding genomes, and can be written out to store for use in subsequent workload stages. The second challenge is choosing an index structure that works well on flash storage. For LMAT to achieve useful performance, it must make use of multicore architectures, in which the application must process many input reads concurrently for classification. Flash has orders of magnitude higher latency than DRAM, requiring the design of latency-tolerant algorithms and data structures for k-mer search. Since our classification algorithm has compute intensive phases in addition to the data-intensive database lookups, the goal is to gain latency-hiding between the two parts of the query application process by running a large workload of query processes in parallel, ideally resulting in higher LMAT application throughput.

This paper focuses on speed performance and scalability aspects of the LMAT workflow. Discussion of accuracy and in depth comparison with other techniques for metagenomic sample analysis can be found in [2]. In developing LMAT, we

experimented with several values of  $k$  in the range  $17 \dots 21$  to maximize classification accuracy and performance. In this paper we report performance primarily with 20-mers, which were previously found to support accurate classification.

The following sections discuss the workflow to create the index, the optimizations to several workflow components, and the evaluation of those optimizations.

## 3. DATABASE PREPARATION

The LMAT classifier depends on a searchable k-mer database. In our implementation, the k-mer database is a key-value store in which the key is a 64-bit binary encoded form of the k-mer and the value is the list of genomes and/or higher level taxonomy classification (strain, species, genus, family, ...) that contain the k-mer, called the taxonomy identifiers (tax IDs). Figures 1(a) and 1(b) contrast two workflows we have used for generating the database. The first phase is k-mer extraction, performed by the program Jellylist in Figure 1(a) and K-mer Prefix Counter in Figure 1(b). The k-mer extraction phase takes a preprocessed genome reference database and enumerates all the canonical k-mers. In canonical form, a k-mer and its reverse complement<sup>1</sup> are considered equivalent, and the k-mer with a lower value internal representation (64-bit ID) is used.

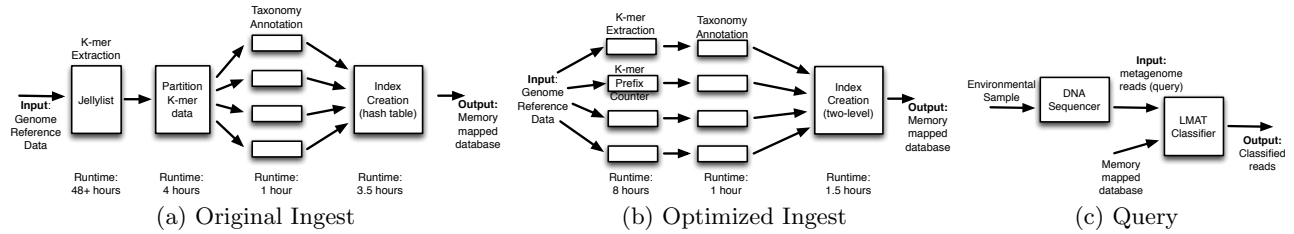
Then, each extracted k-mer is annotated with a list of tax IDs that contain the k-mer. Tax IDs are stored as 32-bit integers. The two workflows accomplish this task in different ways. In the first workflow (Figure 1(a)), the output is so large that it must be partitioned for the Taxonomy Annotation phase. In the optimized workflow of Figure 1(b), the output of k-mer extraction is already partitioned. The partitioned k-mer sets are annotated with the taxonomy lists. Finally, the annotated k-mers are indexed into a searchable form. The final phase of both workflows reduces the many partitions into one single output database.

The workflow makes use of large and fast scratch storage for the intermediate storage of all the files generated by the first two phases. These files can be stored on any conventional file system, though use of a parallel file system is suitable for the workflow stages running parallel tasks. The k-mer extraction phase produces output file(s) roughly 10 times larger than the original input files. An equivalent amount of storage is needed for the partitions, given the workflow in Figure 1(a) where we have a single output from the k-mer extraction phase. Output from the taxonomy annotation phase takes 2.15 times more storage than its input (for  $k = 20$ ). For example, a 19 GB input set, produces 190 GB of files with the extracted k-mers and genome identifiers. Partitioning requires an additional 190 GB. The data size after taxonomy annotation is 400 GB. In total, close to 800 GB of storage is needed for the full reference set.

### 3.1 K-mer extraction

The software program Jellylist was initially selected to do the k-mer extraction phase since it was an existing k-mer analysis tool with an early software variant, Jellyfish [12],

<sup>1</sup>The reverse complement is the reverse of the k-length sequence with each nucleic acid base replaced by its complement:  $A \leftrightarrow T$  and  $C \leftrightarrow G$ .



**Figure 1: LMAT ingest and query workflows.** The original workflow has four stages. The optimized LMAT workflow improves the runtimes of the k-mer extraction (K-mer Prefix Counter vs. Jellylist), database ingest stages, and removes the partitioning stage. Example times assume 19 GB reference sequence input.

which was previously shown to enable fast multithreaded k-mer counting. (The Jellylist code is not published but freely available from the Jellyfish authors.) Jellylist uses lock-free hashing techniques and creates lists of k-mers and genome identifiers using an in-memory data structure. Thus, the application requires a large shared-memory node to process a set of genome reference sequences. Jellylist includes the feature of storing positions each k-mer within the containing genomes. Although we chose not to use this feature, as the positions are not required for LMAT, the feature remains a key part of the application and may impact its performance even when not in use. Unfortunately, Jellylist processing resulted in excessively long runtimes (including the final writing of k-mers to storage). For instance, to extract 20-mers from a 19 GB input reference set took close to 48 hours on a single server with 1 TB DRAM. The addition of the human genome to that input set, 3 additional GiB, added unforeseen complexity, and the run could not complete within our seven day allocation on the server.

The first approach we considered to addressing the problem of intractable Jellylist runtimes was to partition the input reference set of sequences. This approach is problematic as well. Ultimately the partitions need to be merged. The challenge in the merge is accounting for the k-mers extracted from several genomes in multiple partitions. For these repeated k-mers, the merge step must combine the lists of constituent genome identifiers. Moreover, all partitions require sorting in order to facilitate efficient merging. We went through this process to merge the set of k-mers extracted from a human genome with those extracted from our microbial genome data set. The process was long, cumbersome and error prone, making use of both in-core and out-of-core merge steps, and was eventually eliminated from the pipeline.

We instead design an alternative approach to partitioning. Rather than partitioning the input sequences, we partition the k-mers as they are extracted. We call our application `kmerPrefixCounter`. Given that nucleic acids are comprised of four bases, every k-mer has a “prefix” of  $n$  bases, so there are  $4^n$  possible prefixes. We can pick the desired number of partitions. Based on picking a prefix length  $n$  we can determine the number of partitions that fits our resources. For instance if we want to have four partitions,  $n = 1$ , a each partition extracts k-mers of a particular initial base (A, C, G, T). Each process works independently and no inter-process communication is required. For sixteen partitions,

the prefixes are AA, AC ... TT. All partitions’ processes see the same sequence information, but based on the prefix, each process chooses whether or not to include a particular extracted k-mer within its own k-mer set. The output files do not require additional sorting and merging because each partition has a non-overlapping set of k-mers.

Our implementation of `kmerPrefixCounter` uses C++ STL data structures, the `std::map` for recording the k-mers (one per partition) and the `gnu hash_set` for managing each list of genome identifiers per k-mer. It is a very simple, straightforward implementation, albeit does not make optimal use of memory. The execution of `kmerPrefixCounter` creates a set of output files with each file being already sorted by k-mer key, as enabled by the use of the `std::map` data structure. The sorted output becomes important because it needed to create the two-level index, which is optimized for sorted input. Sorted output is also useful because it facilitates merges of sorted sequences if the user chooses to add extracted k-mers from an additional set of organisms at a later date.

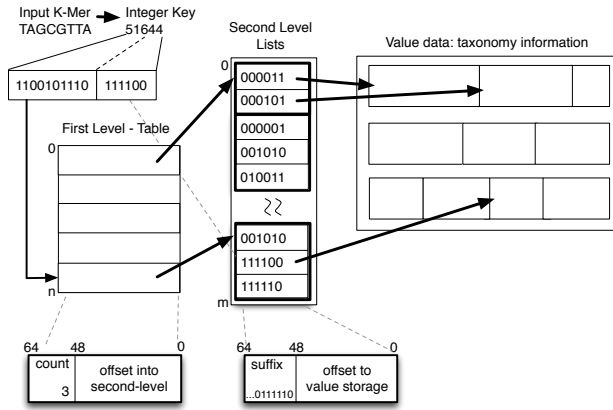
`KmerPrefixCounter` has two modes of operation. In “independent job” mode every partition is started as its own single-threaded process with the input reference sequence file, the prefix value  $n$ , and the partition ID identifies when prefix the process handles. All partitions must read the entire reference sequence file and parse individual k-mers.

In MPI “cluster” mode, the primary rank iteratively reads and broadcasts each sequences in the file. In essence, this shifts the cost of the file I/O performed by all processes in the independent job mode to the MPI broadcast operation. Each MPI task extracts individual k-mers from sequence data.

In practice, independent job mode allows for flexibility in using job schedulers within our cluster computing systems, as it is potentially more difficult to obtain a single allocation for a large group of nodes simultaneously rather than many single node allocations. Our evaluation shows results of both the independent job and MPI modes of `kmerPrefixCounter`.

## 3.2 Taxonomic Annotation

The output from k-mer extractions contains k-mers, each with a list of genome identifiers. The genome identifiers refer to individual sequences rather than particular organisms. In the taxonomic annotation process, we map the genomes to an NCBI taxonomy identifier for the organism. At the



**Figure 2: Design of the two-level index. Each k-mer is split. The prefix is the index of the first level. Lists of the suffixes with offsets into the Value Data store are in the second level.**

level of the genome, this is typically a species or strain identification. K-mers that map to more than one organism have a common taxonomic ancestor within the NCBI taxonomy tree. The second part of the taxonomy annotation computes the lowest common ancestor (LCA). The procedure is embarrassingly parallel, as it is performed on a per-k-mer basis. We process each partition of extracted k-mers and their corresponding genome lists as separate processes. Each writes its own output file. Additionally, this stage allows us to collect counts of total k-mers and taxonomy identifiers for use in sizing the final indexed database.

### 3.3 Index Creation

A primary requirement to running the LMAT classification application is efficient k-mer lookup. The k-mer database index must be organized for very fast lookup as opposed to insert or delete. For the purpose of classification only, the indices will not be updated, as they serve in a write-once, read-many usage model. Thus, use of an off-the-shelf RDBMS database package for the index is not appropriate, and other available key-value stores incur unnecessary transactional overhead.

Given that our lookup requirement for LMAT does not include range queries, we determined that a hash table would be an appropriate fit. The keys specific to our application are 64-bit integer values: the encoded k-mer. To begin rapid prototyping for the LMAT classifier, we selected the gnu hash\_map data structure. We performed some simple performance tests of the data structure in comparison with Google’s sparsehash and C++ std::map (a “baseline” for a tree-based map for a point of comparison) and found that gnu hash had the best performance. We manage the storage of the taxonomy identifier lists in a byte array, written to in an append-only “log-like” manner; thus, their storage is compact without additional compression. Offsets into this byte array are stored as the values of the hash\_map.

The hash table is built in a persistent heap as a memory-mapped file. The use of memory map presents a convenient

programming abstraction for performing out-of-core data access without the overhead of application buffer management and mix of standard I/O and in-core data structure access. To enable the index to be persistent after ingest and be used afterward for classification, we configure gnu hash with a custom memory allocator that is backed by storage. Previous work at LLNL, “perm-je” has modified the Jemalloc memory management library [6] to enable memory allocation from an address range memory-mapped to a file residing on a storage device. Jemalloc is a drop-in replacement for regular malloc routines for allocating memory. Our modification to jemalloc allows for an additional step to specify the database filename (jemalloc memory-maps to temporary files). Our “perm-je” library is available as open source and distributed with LMAT at SourceForge.

To meet the goal of running LMAT using flash storage for indexing, it became necessary to consider an alternative index to gnu hash. Our preliminary work with the index showed poor query performance when reading the data structure from flash. In designing such an index, we focus on the goals of improving page reuse and locality of access.

We have collected some statistics on our use of the gnu hash table. The maximum bucket count hard coded into gnu hash is 4,294,967,291. Thus, on average, regardless of hash function, the chain length for each lookup is 2.15 for the 9.21 billion items of a 19 GB reference sequence set. The chaining becomes problematic when the hash table resides on flash storage: each pointer dereference to traverse the linked lists potentially forces additional page faults. For this reason, we focus on improving locality when handling collisions.

Our alternative data structure to the gnu hash\_map is a “two-level” index. For this approach, we split the 64-bit k-mer into a prefix and suffix. Note that this “prefix” is different from those used in the discussion of kmerPrefixCounter. The first level maps every prefix into an array of the suffixes stored in the second level. This structure resembles a hash table in that the first level is similar to the hash table structure; in our case the “hash function” returns the prefix of the key, which is the index into the first-level array. The second level manages the collisions that occur — as we expect many matching prefixes — by maintaining array-based lists containing the suffix. These lists are maintained in sorted order so they can be quickly binary searched during lookup operations. It is unnecessary to store the entire key (the encoded k-mer) because once the prefix is used within the first level lookup, it is no longer needed. In contrast, chaining within a hash table stores the entire key.

Figure 2 illustrates the structure of the index, where  $n$  is the length of the first-level array (corresponding to selected k-mer prefix size), and  $m$  is the length of the second-level array of lists. Note that  $m$  is equal to the total number of k-mers in the database. In our implementation of the index, both the first and second levels are 64-bit integer arrays. The boxes at the bottom of each table show the fields within each 64-bit value from each of the tables. For each 64-bit value in the first-level array, the 16 most-significant bits store the count of items in the corresponding second-level list of suffixes for k-mers that share that common prefix (the index

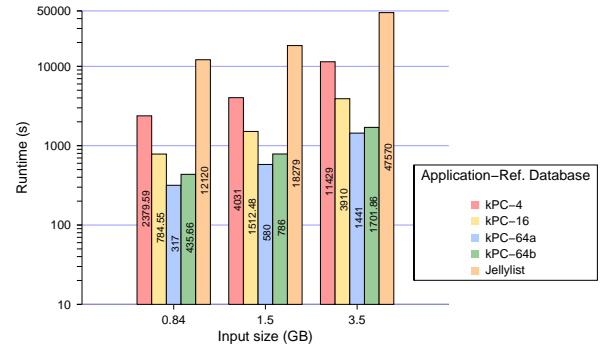
into the first-level array). The remaining least-significant 48 bits store the offset into the second level, pointing to the sorted list of k-mer suffixes. In the second-level array, the k-mer suffix is stored in the most-significant 16 bits of the 64-bit value. The remaining 48 bits of the second-level array value are used to store the offset to find the taxonomy identifier lists in the value data array, equivalent to the gnu hash\_map value field.

Note that this data structure is specific for integer keys. While we apply it in this work to integer-encoded k-mers, it can potentially be applied to other integer-specific workloads. It is not suitable for general purpose key-value storage that typically performs a hash-function calculation on variable-length strings of ascii characters.

An important determination needed to use the index is how to split the input k-mer between the first and second level. The split parameter determines both the size of the first level table and the maximum length of each second level list. The average length of each list and distribution of such lengths depends on the particular integer key data indexed. Our goals in considering specific split parameter are to: (1) try to keep the second level lists on a single page of memory so as to incur fewer page faults; (2) keep the first level table size small enough to remain cached in DRAM. An additional consideration (implementation specific) is that the 16-bit count field limits second level list lengths to integers of that size. Thus, we must balance the tradeoff of optimizing for each of these goals.

We select split factors where the prefix sizes range from 24 to 31 bits with suffixes ranging from 16 down to 9 bits respectively, each pair a total of 40 bits for the k-mer length of 20 base pairs. At one extreme of these parameters, we have a 128 MB table for  $2^{24}$  possible k-mer prefixes with maximum second level list lengths of  $2^{16} - 1$  spanning up to 128 4K pages for the longest lists. Note that this configuration fits the constraint of the largest possible list length based on a 16-bit integer count value. At the other extreme of the range, a 9-bit suffix guarantees that the maximum list fits on a 4K page, given 512 64-bit values. However, the first level is 16 GB ( $2^{31}$  prefixes) using this configuration, which may not remain fully in the mmap handler’s buffer cache when memory is limited. Our evaluation considers LMAT classification performance given this range of settings.

The ingest procedure for the two-level index assumes that input data is sorted by the k-mer key. The output of the kmerPrefixCounter code produces sorted k-mers, which facilitates this process in our workflow. This enables the second level list values to be added in sorted order, so no additional sorting is needed. The ingest procedure is very straightforward. The ingest code initializes the first-level array to zero values. The second-level is written to in a log-like fashion, so the procedure must maintain and increment the offset pointing to the last value written. For each k-mer in the input set, the procedure first splits the k-mer into the prefix and suffix. For the first-level, it checks if the value in the array for that prefix has been written. If not, it writes the current second-level last value offset to the lower 48 bits of the value and sets the count field to an initial value of 1 for that prefix. If the location in the array has been written, the count



**Figure 3: Performance of methods for k-mer extraction using 1 TB node(s).** kPC-64a (kmerPrefixCounter, 64 concurrent partitions) ran on two nodes (32 processes on each) and kPC-64b ran on a single node with hyperthreading enabled (80 hardware threads). All other series ran on single node with hyperthreading disabled. Jellylist set to use 32 threads. The sizes on the x-axis represent collections with 500, 1000 and 2000 individual genome sequences respectively.

Input size GB	Partitions	Runtime (seconds)
3.5	256	600
13	256	2160
19	256	5480
22	256 out of 1024	2604
55	256 out of 1024	7020

**Table 1: kmerPrefixCounter execution time on a compute cluster using MPI mode.** Runs used 256 nodes regardless of the number of requested partitions. In the cases where 1024 partitions were requested, the first 256 partitions (AAAAA-ATTTT) were run on the cluster.

is incremented, but the offset into the second level for that particular prefix has already been set. For each input k-mer, the suffix and offset to taxonomy storage are written to the second-level array at that last-value offset, maintaining the sorted order of suffixes for common prefixes as they are written. To additionally contrast the two-level index with gnu hash (considering ingest), despite the disadvantages of gnu hash that we observe, an advantage of the gnu hash index is that it accepts unsorted k-mer input to build the index.

## 4. EVALUATION

Our evaluation of the LMAT workflow considers three aspects. First, we examine the performance of our cluster k-mer extraction application in comparison to the Jellylist application. Second we profile the build of the index, comparing the performance of gnu hash\_map with the two-level approach. Third, we compare the two index techniques when used for query in LMAT classification and examine the performance of the two-level index under varying amounts of memory and configurations.

Single node measurements are run on a 4 socket 2 GHz Intel E7 4850 CPU with 1TB of memory. Index creation and database query experiments use this hardware. K-mer extraction is evaluated on large memory nodes and on conventional cluster nodes. The latter are Infiniband-connected 2.6 GHz Intel E5-2670 CPUs with 32 GB of DRAM each. All query experiments use PCIe attached flash memory. We use a software RAID with 2 FusionIO 1.2 TB ioDrive cards. Our experiments run in a standard supercomputing center environment in which input and output files are read from or written to a 1.5 PB Lustre file system.

## 4.1 K-mer extraction

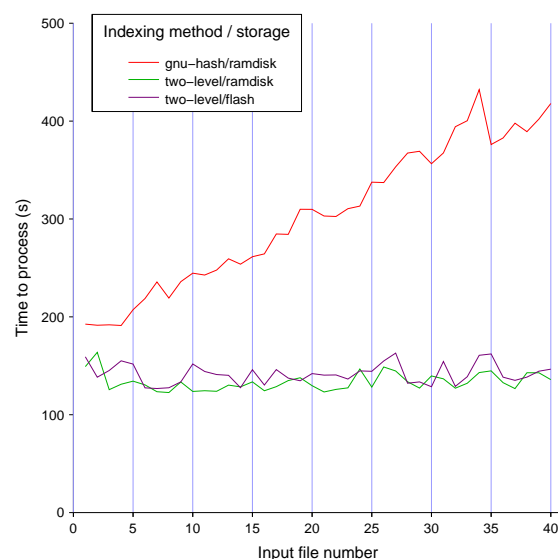
Figure 3 plots the runtime for k-mer extraction comparing Jellylist with the kmerPrefixCounter (labeled kPC). These kmerPrefixCounter runs do not use MPI; thus, each partition runs as its own process and opens the same input file as all others. We consider three input sizes (X-axis) and for kmerPrefixCounter, four configurations. The three input sizes represent collections with 500, 1000 and 2000 individual genome sequences respectively. The number following the label indicates the number of partitions run concurrently. For 64 partitions, we consider two variations: (kPC-64a) use of two nodes with all processes running on physical cores or (kPC-64b) use of a single node with hyperthreading enabled; with hyperthreading the 64 processes run concurrently on the 80 available hardware threads (with 40 underlying physical cores). The results show that using the kPC-64b configuration (single node with hyperthreading) incurs an 18% performance penalty (for the largest workload shown) over kPC-64a, which uses 64 CPUs (on two nodes) without hyperthreading. In other words, using a CPU per process reduces runtime by only 18% over the use of the hardware threads with 40 physical cores.

In all, we observe a range of speedups of  $4.17\times$  for kmerPrefixCounter over Jellylist when using four processes and  $27\times$  when using 64 processes and hyperthreading enabled. Experience with Jellylist has shown that hyperthreading does not improve the application’s performance.

A single 1 TB node has sufficient memory to extract k-mers from a 22 GB input set, including the human genome. The runtime took 8.12 hours to complete using 64 partitions with hyperthreading enabled.

Use of a cluster allows for a greater partitioning factor than on a single large memory node. For input data sizes up to 19 GB (does not include human genome), we run with 256 partitions on 256 nodes. All these runs complete in under an hour, as shown in Table 1. For the larger two sizes shown, 22 and 55 GB, we could not complete runs with 256 partitions because the partitions that required the most memory would not fit on the 32 GB node. Instead, we increase to 1024 partitions. Timings are shown for subsets of the entire workload<sup>2</sup> with 256 nodes (one quarter). These took the first (AAAAA - ATTTT) partitions. These partitions have the largest numbers of k-mers and taxonomy IDs, thus requiring the most memory and runtime. We also ran the 55GB dataset with the last (TTTTT) partition, which

<sup>2</sup>Due to cluster job scheduling policies, it was not possible to reserve a single 1024 node allocation, and therefore we queued several smaller allocations independently.



**Figure 4: Comparison of indexing approaches measuring ingest timings. For the two-level index we consider performance of both ramdisk and flash. Gnu hash performance on flash was too poor for consideration. Each point on the x-axis represents a roughly equal number of k-mers along the x-axis.**

generated 193 thousand k-mers vs 109 million for the first (AAAAA) partition. Its runtime was 99 minutes, only 18 minutes shorter than the first, indicating that for this partitioning, I/O and parsing dominated the runtime rather than adding k-mers to the map data structure and writing out the final list of k-mers and taxonomy IDs.

For practical reasons, we found it most expedient to run partitions that required more than 32 GB on large memory nodes and the remaining partitions as independent jobs on the conventional cluster. This entire process takes about 8 hours.

## 4.2 Index Creation

We discuss the performance of the ingest process that builds the index from files containing k-mers and taxonomy information, comparing use of gnu hash with the two-level index. Figure 4 shows timings of the processing of input files on a 1 TB node. For this experiment we use input data derived from 22 GB data set, which includes microbial organisms and a human genome. Each partition (x-axis) is a single file, containing roughly the same number of k-mers, and they are processed sequentially. The ingest application measures the time to process each input file. For the gnu hash index, the time to ingest each successive partition increases with elapsed runtime. We attribute the increase to the growth of chaining as the hash table fills up. In contrast, the processing time for each partition file for the two-level varies slightly between each, but does not appear to progressively increase. The total times (approximate) for gnu hash and two level hash are respectively 3.5 hours and 1.5 hours. For the two-level index, we have included timings from both a memory mapped ramdisk and memory-mapped flash array



with memory available for the application and buffer cache limited to 16 GB. The overhead for using flash over ramdisk in this case is 6.2%. This small overhead suggests that the working set size for the tow-level index ingest is smaller than 16 GB.

Previous experiments with the gnu hash\_map ingest have shown that using flash storage and a limited amount of memory for buffer cache produces considerable increases in time. Therefore, we choose not to show those as well, as they are not comparable and would not complete in a reasonable time for the data set we use in this experiment.

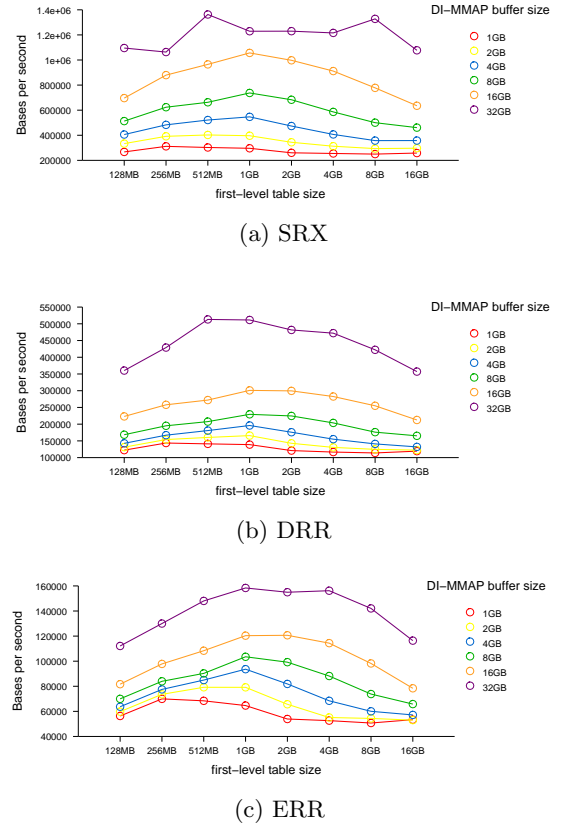
For this data set, the total storage used by the gnu hash\_map index was 522 GB, including the taxonomy information. In contrast, the two-level index used 320 GB. Based on the the size of the hash table, we have measured 27.3 bytes per k-mer. In contrast, the two level hash uses 8.012 to 9.53 bytes per k-mer for this workload. The two-level hash is fixed to 8 bytes per k-mer from the second level plus the memory overhead of the first level, whose size varies depending on the split parameter. Thus, as the number of k-mers increases, that overhead becomes a less significant part of the total storage. Given that the two-level index is more efficient in its use of storage, we expect it to have better page reuse than gnu hash when queried off of a flash device.

### 4.3 Index Lookup Performance

We examine the performance of LMAT classification using flash memory as a store, in which we evaluate the two-level index in two ways. First, we compare the performance of eight index configuration. Each configuration has a different split between prefix and suffix in the k-mer key, resulting in varying first-level table sizes and second-level list lengths. Second, we compare the query performance of the two-level table with gnu hash.

These experiments make use of the DI-MMAP custom memory map handler [17] to access the memory-mapped database files. The goal of DI-MMAP was to address the performance gap in standard linux memory-map for large memory mapped files. The key features of the runtime are a fixed size page buffer, whose size is a configuration parameter; minimal dynamic memory allocation; a simple FIFO buffer replacement policy; and preferential caching for frequently accessed pages. DI-MMAP superior performance to regular system mmap is measured up to  $4.88\times$  as shown in [17]. Our experiments consider 6 configurations of the DI-MMAP buffer size from 1 GB to 32 GB.

For these experiments we consider three non-synthetic metagenomic samples representing a viral metagenome (SRX022172) (0.5 GB), a human microbiome metagenome (ERR011121) (2.5 GB) and a single species raw read 'metagenome' (DRR000184) (0.6 GB), taken from the NCBI sequence read archive for our input sets (each are abbreviated to the initial three letters). These experiments run the LMAT software V1.1 adapted to run with the two-level index using a 11.2 billion k-mer index from five kingdoms or domains of microorganisms and the human genome. The results of these experiments measure the input base-pairs per second (correlates with input bytes per second) processed by the LMAT classification application, shown on the y-axis of Figures 5 and 6. All the



**Figure 5: Index configuration performance selecting various first-level table sizes with varying DI-MMAP buffer sizes. The table sizes are selected based on the range of k-mer split parameters considered in section 3.3. Each subfigure is a different input set taken from a non-synthetic metagenome, each with different redundancy and taxonomic diversity characteristics.**

experimental runs for LMAT classification use a single 40-core node running 160 concurrent threads.

Figure 5 show the performance with the eight different index configurations based on the k-mer prefix-suffix split range we identified in Section 3.3. We use the first-level table sizes based on those splits for the x-axis because those sizes can be compared with the various DI-MMAP buffer sizes that each series represents. Each subfigure uses input data from the SRX, DRR, and ERR data sets, respectively. Note that the performance differences among the three sets varies considerably. This is due to different amounts of redundancy affecting hit rate and taxonomic diversity affecting classification performance. The pattern we observe with most of the curves plotted within the figure is that performance improves from the 128 MB first-level table size to 1 GB size, then declines to the 16 GB size. Under the conditions of the smaller size, the entire table fits comfortably into the DI-MMAP buffer, yet the second level lists are longer on average.

We note several exceptions to the pattern observed above. One difference among the three input sets is that the SRX

Metagenome	gnu hash (Kbp/s)	two-level (Kbp/s)
SRX	322	2580
DRR	756	1710
ERR	1220	3220

**Table 2: LMAT classification performance of gnu hash and two-level index, each using a ramdisk for reference database storage. Each row gives the input metagenome. Values listed in the second two columns are in thousands of base-pairs per second (Kbp/s).**

data set with 32 GB buffer does not have the same single peak pattern. We have observed that the SRX data set has a relatively high number of redundant k-mers. In contrast to the other data sets, the hit rate becomes extremely high for this data set with the 32GB buffer. Additionally, with a 1 GB buffer, the 1 GB table size is not the peak performer. This result we expect because the 1 GB table cannot fit entirely in the buffer without its pages being evicted when requests are made for pages from the other components of the index.

Figure 6 shows the performance of LMAT using the two-level index compared with the gnu hash\_map index. We include three index configurations for comparison. The x-axis in each plot is the DI-MMAP buffer size. As correlating with the previous data, the buffer size has considerable impact on the performance of the two-level index. However, due to the mostly random-access pattern of the gnu hash\_map, increasing the buffer has little impact. Looking at each subfigure for the three input data set and the optimum configuration of the two-level, we observe speedups of  $74\times$  for SRX,  $13.7\times$  for DRR and  $8.4\times$  for ERR over the gnu hash\_map index.

We present performance of gnu hash and the two-level index when performing LMAT classification using ramdisk to store the database in Table 2. While not as large a difference in performance between the two index approaches, the two-level index outperforms gnu hash by approximately  $2.2\times$  to  $8\times$ . Additionally, in comparing two-level index performance on ramdisk with the top performance measured in Figure 6, for the SRX input set, performance using flash is approximately  $2.1\times$  slower than ramdisk, but at most  $20\times$  slower when measuring the ERR input set. The ERR has lower taxonomic diversity but less redundancy; thus, when running with the index stored on the ramdisk, it achieves much higher performance with the lower latency. The higher diversity in SRX makes classification (aside from index lookup) take longer, so moving to lower-latency storage does not make as great a difference.

## 4.4 Discussion

The key finding for the k-mer extraction experiments is that partitioning is necessary for improving performance, whether it be on a single large-memory node or cluster. Our results show that the prefix-oriented partitioning strategy significantly reduces runtime for concurrent processing. However, memory requirements appear to increase with the input size. Thus, either larger memory nodes or larger cluster allocations are needed, as we expect to see larger in-

puts as more organism are sequences and incorporated into genome reference sets. I/O requirements of kmerPrefixCounter suggest a need to optimize our handling of I/O, e.g. using parallel I/O.

Two-level ingest directly to a node with flash storage appears to be a viable alternative to ingest using a large-memory node. While there is not one clear choice for how to configure the two-level index for query, we observe that configuring the index within the range of 26 - 28 bits for k-mer prefix appears to be favorable, and the extremes of the range do not.

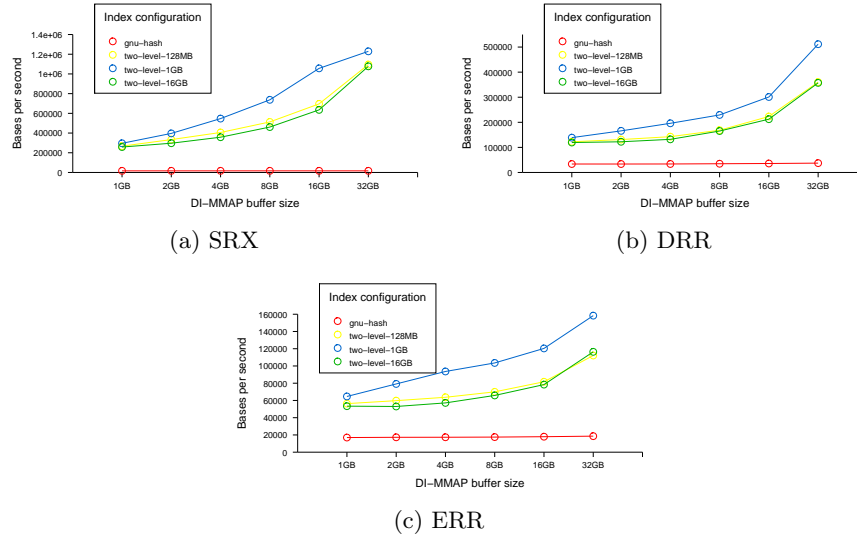
## 5. RELATED WORK

For metagenomic analysis, two approaches applied toward making search more scalable include reference database size reduction and faster database search methods. Reference database size reduction is achieved through the use of genetic markers storing only the more informative sequences [8]. Genometa features a user interface to enable classification using its own scheme for read mapping [4]. MetaPhlAn uses a reduced database set and can use either bowtie or BLAST to map reads to its input database prior to taxonomic classification[15]. Marker based approaches offer efficient summarization of metagenomic contents, but only cover a portion of the query set leaving novel and other informative reads buried within the larger pool of unclassified reads, which could require additional examination [13]. A less lossy approach reduces sequence redundancy by storing only the genetic differences among reference genomes. This approach was shown to speed up BLAST and BLAT genome database searches [9].

Faster database search methods apply large search seeds and examples include BLAT [16], BWA [5] and other read mapping tools [11], but analyzing the search results remains a challenge with some approaches selecting the lowest common ancestor of multiple matches and others using variants of a best match selection procedure to improve rank specificity of the reported taxonomic label. Moreover, parameter settings of the search tools can dramatically alter the outcome of the reported label and must be considered carefully [10].

Other projects have taken approaches to read mapping and scaled them to cluster computing. Note that the cluster approaches alone do not attempt metagenomic classification. mpiBLAST supports the mapping of reads utilizing cluster computing resources. It partitions its reference genomes database by creating what the authors refer to as database fragments and also partitions the input query read into multiple segments. mpiBLAST-PIO features several parallel I/O optimizations, namely, it offloads the formatting and writing of results from a master process to the workers. This feature increases the scalability of mpiBLAST to hundreds of thousands of processors. [3]

CloudBurst makes use of the Hadoop framework to scale read mapping to a cluster. CloudBurst uses a k-mer based approach — well-known seed-and-extend algorithms — but unlike LMAT, it does not index every k-mer in the reference database[14] Crossbow is another Hadoop-based cluster read mapper, which is based on using bowtie as the kernel that



**Figure 6: LMAT classification performance comparing the two-level index with gnu hash-map.** For the index configurations, the size indicated after two-level gives the first-level table size for the index: the two extremes of the range of values considered and one median value. Each subfigure is a different input set taken from a non-synthetic metagenome, each with different redundancy and taxonomic diversity characteristics.

maps the input reads against the reference sequences[7].

## 6. CONCLUSION AND FUTURE WORK

This paper demonstrates a scalable approach to generating and querying very large searchable databases, starting with reference sequences, for use with the LMAT classifier. We have evaluated the alternative approaches to the database ingest pipeline. This evaluation includes a comparison of k-mer extraction using a conventional cluster or a single large memory node. To facilitate search, we presented the design of a two-level index data structure uniquely tuned for flash storage and demonstrate its speed: an improvement of 8-74 times versus the use of a conventional hash table.

We can identify several improvements to make to kmerPrefixCounter running as an MPI cluster application. First, we need to load-balance the partitions. The higher prefix partitions may be combined to make better use of memory. Second, we can consider an alternative input strategy where the input genome reference sequences are partitioned, and the k-mers would be routed to the appropriate destination partition.

A potential area to explore for LMAT classification is a distributed index on a cluster. A naive partitioning based on k-mer prefix, using the partitions from kmerPrefixCounter would result in hotspots; thus, we need to do more intelligent partitioning and routing of input k-mers, given that ideally, we would split up the input reference sequences in order to alleviate the cost of processing all the reference sequences sequentially when stored in a single file.

We propose two potential improvements to the two-level index. First would be to investigate an ingest procedure for two-level indexing that allows for k-mers inserted in sorted

order, and this may alter the data structures as well while trying to maintain the same underlying principle for query. Second would be to examine a potential parallel ingest of the k-mer data into the two-level index.

## 7. ACKNOWLEDGEMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded in part by Laboratory Directed Research and Development grants 33-ER-2012 and 08-ER-2011, and DOE Office of Science grant KJ0402000-SCW1076.

## 8. REFERENCES

- [1] J. E. Allen, S. Ames, D. Hysom, S. Garnder, and G. S. Lloyd. Lmat: Efficient taxonomic labeling of very large metagenomic datasets. <http://http://sourceforge.net/projects/lmat/>, 2013.
- [2] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. LLoyd, M. B. Gokhale, and J. E. Allen. Scalable metagenomic taxonomy classification using a reference genome database. Submitted to Bioinformatics, under review. Available upon request, 2013.
- [3] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiblast. In *In Proceedings of ClusterWorld 2003*, 2003.
- [4] C. F. Davenport, J. Neugebauer, N. Beckmann, B. Friedrich, B. Kameri, S. Kokott, M. Paetow, B. Siekmann, M. Wieding-Drewes, M. Wienhöfer, S. Wolf, B. Tümmler, V. Ahlers, and F. Sprengel. Genometa - a fast and accurate classifier for short metagenomic shotgun reads. *PLoS ONE*, 7(8):e41224, 08 2012.
- [5] C. F. Davenport, J. Neugebauer, N. Beckmann, B. Friedrich, B. Kameri, S. Kokott, M. Paetow,

- B. Siekmann, M. Wieding-Drewes, M. Wienhöfer, S. Wolf, B. Tümmler, V. Ahlers, and F. Sprengel. Genometa - a fast and accurate classifier for short metagenomic shotgun reads. *PLoS ONE*, 7(8):e41224, 08 2012.
- [6] J. Evans. A scalable concurrent malloc(3) implementation for freebsd. In *BSDCan - The Technical BSD Conference*, 2006.
- [7] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome biology*, 10(11):R134+, Nov. 2009.
- [8] B. Liu, T. Gibbons, M. Ghodsi, T. Treangen, and M. Pop. Accurate and fast estimation of taxonomic profiles from metagenomic shotgun sequences. *BMC Genomics*, 12(Suppl 2):S4, 2011.
- [9] P.-R. Loh, M. Baym, and B. Berger. Compressive genomics. *Nat Biotech*, 30(7):627–630, 2012.
- [10] S. S. Mande, M. H. Mohammed, and T. S. Ghosh. Classification of metagenomic sequences: methods and challenges. *Briefings in Bioinformatics*, 2012.
- [11] J. Martin, S. Sykes, S. Young, K. Kota, R. Sanka, N. Sheth, J. Orvis, E. Sodergren, Z. Wang, G. M. Weinstock, and M. Mitreva. Optimizing read mapping to reference genomes to determine composition and species prevalence in microbial communities. *PLoS ONE*, 7(6):e36427, 06 2012.
- [12] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.
- [13] M. H. Mohammed, T. S. Ghosh, N. K. Singh, and S. S. Mande. SPHINX—an algorithm for taxonomic binning of metagenomic sequences. *Bioinformatics*, 27(1):22–30, 2011.
- [14] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.
- [15] N. Segata, L. Waldron, A. Ballarini, V. Narasimhan, O. Jousson, and C. Huttenhower. Metagenomic microbial community profiling using unique clade-specific marker genes. *Nat Meth*, 9(8):811–814, 2012.
- [16] V. K. Sharma, N. Kumar, T. Prakash, and T. D. Taylor. Fast and accurate taxonomic assignments of metagenomic sequences using metabin. *PLoS ONE*, 7(4):e34030, 04 2012.
- [17] B. Van Essen, H. Hsieh, S. Ames, and M. Gokhale. DI-MMAP: A high performance memory-map runtime for data-intensive applications. In *International Workshop on Data-Intensive Scalable Computing Systems (DISCS-2012)*, Nov. 2012.